# Spring家族介绍及开发环境搭建

## ****Spring能做什么？****

Spring是一个轻量级的控制反转（IoC/DI）和面向切面编程(AOP) 的对象容器框架

Spring官网：http://spring.io/

Spring 是一个于 2003 年兴起的一个轻量级的 Java 开源开发框架

如今Spring已是Java项目的标配，它极大的简化了开发过程，降低了开发难度。

1. 方便解耦，简化开发；
2. AOP 编程的支持；
3. 声明式事务的支持；
4. 方便程序的测试；
5. 方便集成各种优秀框架。

IoC

Inversion of Control

AOP

Aspect Oriented Programming

## ****Spring5新特性****

Spring5 是一个重要的版本，距离SpringFramework4差不多四年

* 支持JDK8
  + Spring 接口中的默认方法
  + 基于 Java8 反射增强的内部代码改进
  + 在框架代码中使用函数式编程 - lambda表达式 和 stream流
* 响应式编程支持Spring Web Reactive异步的、非阻塞的、事件驱动的服务
* 支持J2EE7
  + Servlet 3.1
  + JMS 2.0
  + JPA 2.1
  + JAX-RS 2.0
  + Bean Validation 1.1
* Hibernate 5
* Jackson 2.6
* EhCache 2.10
* JUnit 5
* Tiles 3
* Kotlin

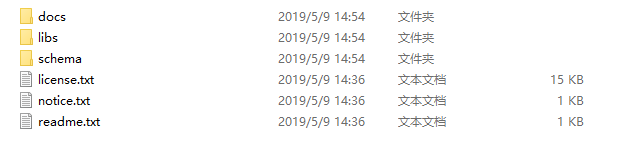
## ****开发环境****

### ****Spring Jar包****

**SpringFramework官网下载地址：**

<https://repo.spring.io/libs-release-local/org/springframework/spring/>

解压缩Jar包可以看见如下结构



1. Doc  文档
2. Libs Jar包
3. Schema XML约束
4. license.txt 许可协议
5. notice.txt 注意事项
6. readme.txt 读我

**常用Jar包**

|  |  |
| --- | --- |
| spring-core | 框架的基础功能，包括IOC和AOP功能 |
| spring-aspects | 提供了与AspectJ的集成，AspectJ是一个面向切面的框架，它扩展了Java语言。AspectJ定义了AOP语法，它有一个专门的编译器用来生成遵守Java字节编码规范的Class文件。 |
| spring-beans | 所有应用都要用到，包含访问配置文件、创建和管理 bean 以及进行 Inversion of Control(控制反转) / Dependency Injection（依赖注入）操作相关的所有类。外部依赖 spring-core |
| spring-context |  |
| spring-aop、spring-instrument | 面向切面编程、植入代理 |
| spring-expression | 模块提供了强大的表达式语言去支持查询和操作运行时对象图。这是对JSP 2.1规范中规定的统一表达式语言的扩展。该语言支持设置和获取属性值，属性分配，方法调用，访问数组，集合和索引器的内容，逻辑和算术运算，变量命名以及从Spring的IoC容器中以名称检索对象。 它还支持列表投影和选择以及常见的列表聚合。 |
| spring-messaging | 消息传递 |
| spring-jdbc、spring-jms、spring-orm | 数据访问支持 |
| spring-jcl | Jakarta Commons Logging采用了设计模式中的“适配器模式”，它对外提供统一的接口，然后在适配类中将对日志的操作委托给具体的日志框架。 |
| spring-tx | 事务 |
| spring-webmvc、spring-web | Webmvc框架支持 |
| spring-webflux | Servlet3.1 + Netty 方式的WebMvc |
| spring-websocket | 对ws支持 |

**commons-logging**

运行Spring程序额外还需要commons-logging包

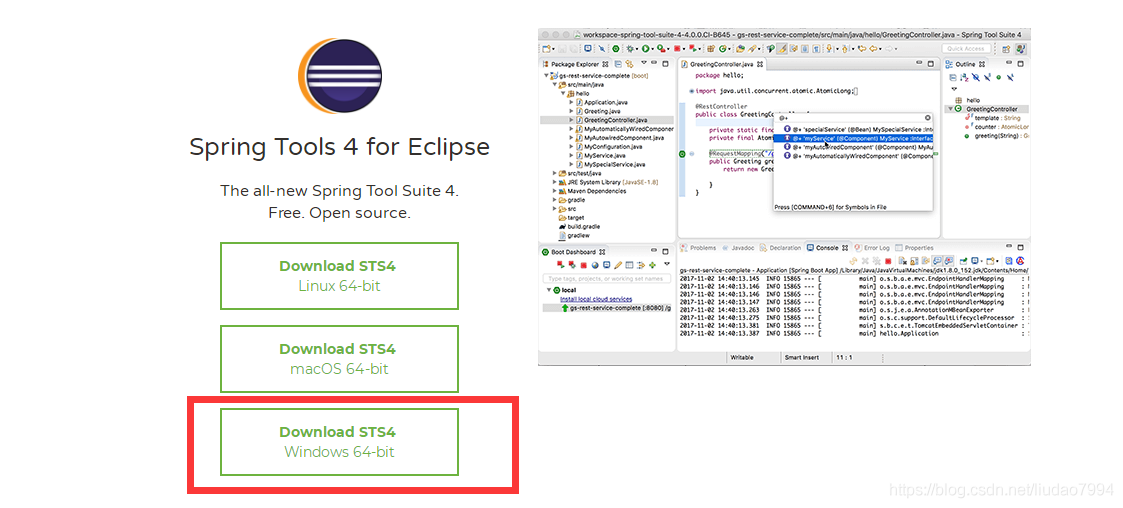
http://commons.apache.org/proper/commons-logging/download\_logging.cgi

### ****开发工具****

可以使用Eclipse、Idea等

推荐使用官方开发工具STS

<https://spring.io/tools>



### ****环境变量****

**window下配置**

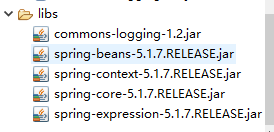
1. 新建->变量名"JAVA\_HOME"，变量值"C:\Java\jdk1.8.0\_05"（即JDK的安装路径）
2. 编辑->变量名"Path"，在原变量值的最后面加上“;%JAVA\_HOME%\bin;%JAVA\_HOME%\jre\bin”
3. 新建->变量名“CLASSPATH”,变量值“.;%JAVA\_HOME%\lib;%JAVA\_HOME%\lib\dt.jar;%JAVA\_HOME%\lib\tools.jar”

在控制台分别输入java，javac，java -version 命令，测试环境是否设置成功

## ****Hello world****

**新建java项目**

**包引入**



**配置文件**

在src下新建

applicationContext.xml

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xmlns:mvc="http://www.springframework.org/schema/mvc"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd">

        "src/applicationContext.xml"

        <bean id="user" class="com.msb.User"></bean>

</beans>

**Bean**

建立准备由Spring管理的Bean

包含两个属性，生成get/set方法

**private** String name;

**private** Integer age;

**测试类**

            ApplicationContext ctx = **new** ClassPathXmlApplicationContext("applicationContext.xml");

            User user = (User)ctx.getBean("user");

            user.setName("uu");

            user.setAge(18);

            System.**out**.println(ToStringBuilder.reflectionToString(user));;

### ****ToStringBuilder 工具类****

1. Commons项目中用来处理Java基本对象方法的工具类包，可以简化很多平时经常要用到的写法，例如判断字符串是否为空等等。
2. 是一个最常用的工具，作为jdk的补充。
3. Lang下有很多Utils类，提供了若干static方法供调用，涵盖了字符串操作、字符操作、JVM交互操作、归类、异常和位域校验等等。

Spring是一个对象管理容器，自动化完成了创建、初始化、销毁等工作。

传统的方式

Person person = **new** Person();

person.setAge(18);

person.setName("zhangsan");

# Spring IOC

## IoC概念

IoC是一个概念，是一种思想，其实现方式多种多样。当前比较流行的实现方式之一是DI。

主要内容：理解IOC 、基于xml的注入、Bean的作用域、单例问题、循环引用问题

ApplicationContext.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

## XML文件结构

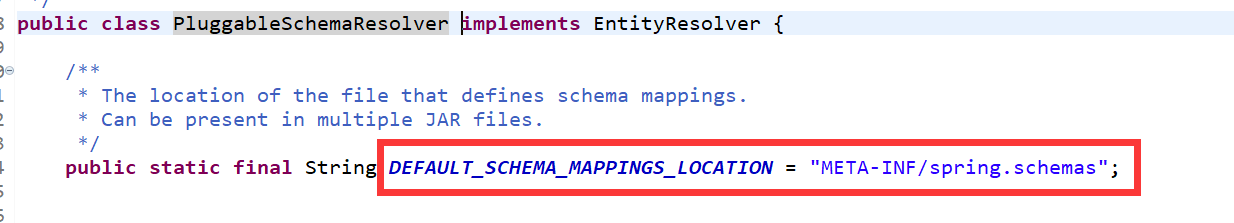
* <beans  beans是xml文件的根节点
* xmlns=[*http://www.springframework.org/schema/beans*](http://www.springframework.org/schema/beans)xmlns=xml NameSpace 类似于java中的package
* xmlns:xsi=[*http://www.w3.org/2001/XMLSchema-instance*](http://www.w3.org/2001/XMLSchema-instance)xsi是指xml文件遵守xml规范，xsi全名：xml schema instance
* xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"*> 是指具体用到的schema资源

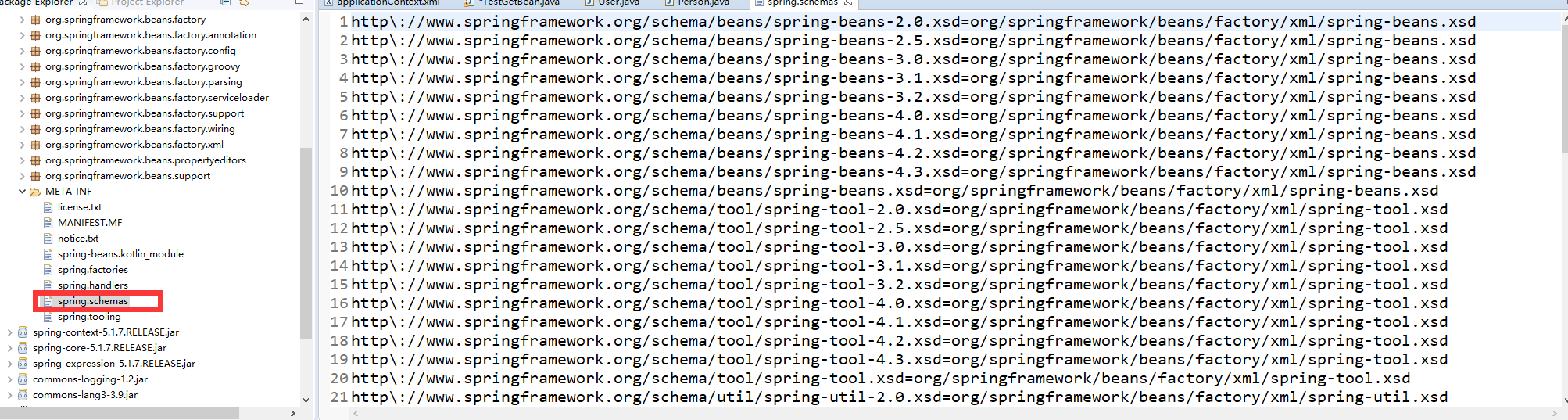
### 真的去网上找xsd文件？

当然不是

spring在加载xsd文件时总是先试图在本地查找xsd文件(spring的jar包中已经包含了所有版本的xsd文件)，如果没有找到，才会转向去URL指定的路径下载

验证PluggableSchemaResolver.class中





=后面是包名以及具体xsd文件位置

### 多配置文件

#### ApplicationContext加载多文件

**new** ClassPathXmlApplicationContext("applicationContext.xml","application-service.xml");

#### 引入外部文件

<import resource="application-service.xml"/>

## Bean的定义与注册

Spring的配置文件是用于指导Spring工厂进行Bean的生产、依赖关系注入及Bean实例分发的“图纸”，它是一个或多个标准的XML文档

<bean id=*"food"* class=*"com.msb.Food"*></bean>

一个bean只能有一个id，但是可以有多个name作为别名

### Alias 别名

<alias name=*"user"* alias=*"my\_user\_bean"* />

## spring ioc container

spring ioc container 管理一个或多个bean，bean来自xml中对bean定义的元数据(configuration metadata)

元数据信息

|  |  |
| --- | --- |
| Class | 类 |
| Name,id | 标识 |
| Scope | 作用域 |
| Constructor arguments | 构造器注入 |
| Properties | 属性注入 |
| autowiring mode | 自动装配 |
| lazy-initialization mode | 懒加载 |
| initialization method | 初始化 |
| destruction method | 销毁 |

### 构造器注入 constructor-arg

Person的构造器

**public** Person(String name, Integer age, Food food) {

**super**();

**this**.name = name;

**this**.age = age;

**this**.food = food;

}

#### Xml

#### 指定name

<bean id=*"person"* class=*"com.msb.Person"*>

<constructor-arg name=*"age"* value=*"18"*></constructor-arg>

<constructor-arg name=*"name"* value=*"zhangsan"*></constructor-arg>

</bean>

#### 指定类型

<constructor-arg type="int" value="7500000"/>

#### 指定index

<constructor-arg index="0" value="7500000"/>

### 属性注入

<bean id="person" class="com.msb.Person">

<property name="age" value="19"></property>

<property name="name" value="zhangsan"></property>

</bean>

#### 使用p-namespace

属性注入

添加一个namespace

xmlns:p=<http://www.springframework.org/schema/p>

使用 p

<bean id="person" class="com.msb.Person" p:age="21" p:name = "zhangsan">

<bean id=*"person"* class=*"com.msb.Person"* p:age=*"21"* p:name = *"zhangsan"* p:food-ref=*"food"*>

#### 使用c- namespace

构造器注入

<bean id="foo" class="x.y.Foo" c:bar-ref="bar" c:baz-ref="baz" c:email= "foo@bar.com"/>

#### 使用java.util.Properties

在set方法中把properties

**private** Properties properties;

**public** **void** setProperties(Properties properties) {

**this**.properties = properties;

**this**.name=properties.getProperty("name");

}

### 对其他Bean的引用

<property name=*"food"* ref=*"food"*></property>

<bean id=*"food"* class=*"com.msb.Food"*></bean>

### 集合

#### Array

private String[] myArray;

<property name="myArray">

<array>

<value>北京</value>

<value>上海</value>

</array>

</property>

#### Properties

**private** Properties adminEmails;

<property name=*"adminEmails"*>

<props>

<prop key=*"administrator"*>administrator@example.org</prop>

<prop key=*"support"*>support@example.org</prop>

<prop key=*"development"*>development@example.org</prop>

</props>

</property>

#### List

**private** List someList;

<property name=*"someList"*>

<list>

<value>apple</value>

<value>orange</value>

</list>

</property>

#### Map

**private** Map someMap;

<property name=*"someMap"*>

<map>

<entry key=*"an entry"* value=*"just some string"*/>

<entry key =*"a ref"* value-ref=*"food"*/>

</map>

</property>

### depends-on 提前初始化

可以使某个bean在创建前，先创建别的bean

### lazy-init

在容器启动后，bean被使用到的时候才加载。可以使用的lazy-init属性

bean id="person" class="com.msb.Person" lazy-init="false"

## 作用域

spring为bean提供了6种作用域，其中4种只有在web-aware的ApplicationContext种才有用。用户也可以创建自定义的作用域。

singleton 、prototype 、websocket、request、session、application

### singleton scope 单例作用域

每一个类，在一个容器内只能产生一个实例

### prototype scope 原型作用域

该bean每次被注入，或者使用getBean()方法获取时，都返回一个新的实例。

### Request scope

该作用域的bean，在每个HTTP request都会新建一个实例，当一个request结束后，该实例也会被丢弃。

### Session scope

某一个用户在一段时间内，会使用同一个session，session有超时时间，过了超时时间则session失效。不同用户使用不同的session。

### Application scope

该作用域的bean，每一个application会创建一个

## MVC下Spring的单例

想在一个singleton内多次调用短存活时间的bean（propotype、request、session等），希望调用的是不同的实例，那么就需要使用AOP proxy技术

### 线程安全问题

业务对象并没有做线程的并发限制，因此不会出现各个线程之间的等待问题，或是死锁问题

MVC中的pojo等实体bean、不是单例的，不被Spring管理生命周期；

### 成员变量

在并发访问的时候这些成员变量将会是并发线程中的共享对象，也是影响线程安全的重要因素

### 引用类型的成员

其中引用类型的成员变量即我们在controller中注入的service，在service中注入的dao，这里将其定义为成员变量主

要是为了实例化进而调用里面的业务方法，在这些类中一般不会有全局变量，因此只要我们的业务方法不含有独立的

全局变量即使是被多线程共享，也是线程安全的。

Controller service dao 层中的业务类是多线程共享的，但是每个线程在处理数据的时候具体处理的数据是在每个线程中各自有一份。

### controller层

* final类型 线程安全
* 成员变量 有状态数据有线程安全问题

## 循环依赖的bean

### 构造器注入循环依赖

作业：

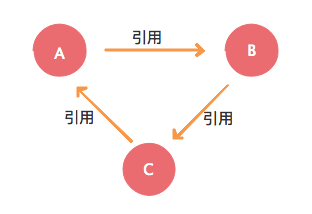
创建3个类

循环引用属性

A类中 -> B类的一引用 -

当循环依赖的bean都是通过构造器注入依赖的时候，无论这些bean是singleton还是prototype，在获取bean的时候都会失败。

### 通过属性注入

* 循环依赖的bean都是singleton 成功
* 循环依赖的bean都是prototype 失败
* 同时有singleton和prototype 当先获取的那个bean是singleton时，就会成功，否则失败
* 

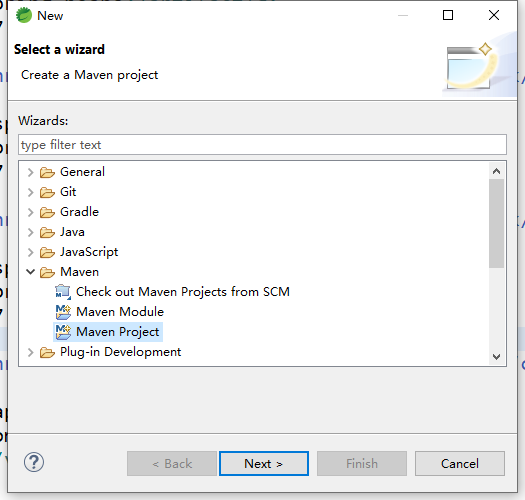
当Spring容器在创建A时，会发现其引用了B，从而会先去创建B。同样的，创建B时，会先去创建C，而创建C时，又先去创建A。最后A、B、C之间互相等待，谁都没法创建成功；

注：单例模式的对象，在Spring中直接实例化，prototype的会检查引用。如果有引用对象没被实例化，暂停实例化，去实例化引用对象；

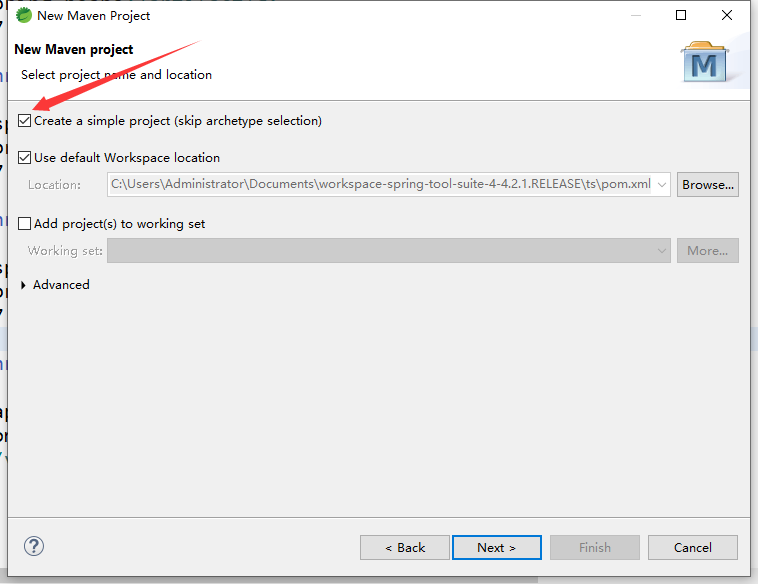
## Maven方式创建Spring工程

### 工程创建

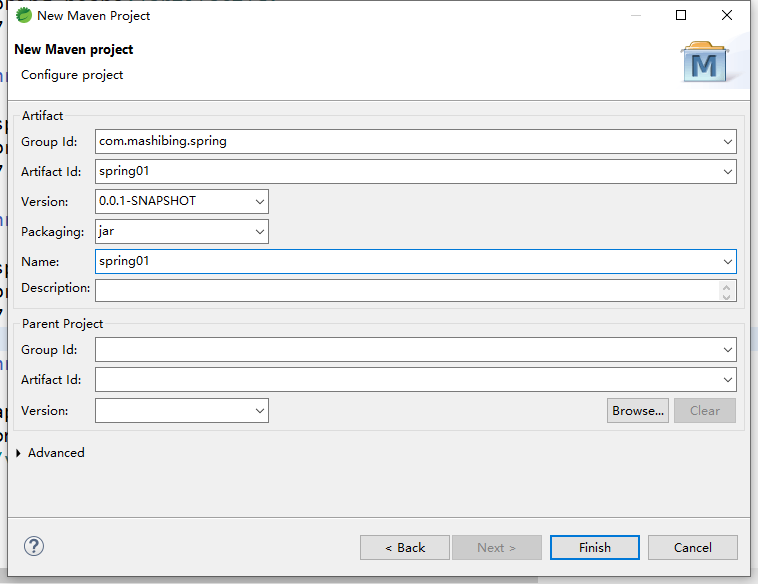
1.新建项目 选择Maven Project



2.勾选 Create a simple project



3.添加项目信息



* Group id ：包名
* Artifact id：标识名
* Name：项目名

### 依赖引入

#### Maven 中央仓库

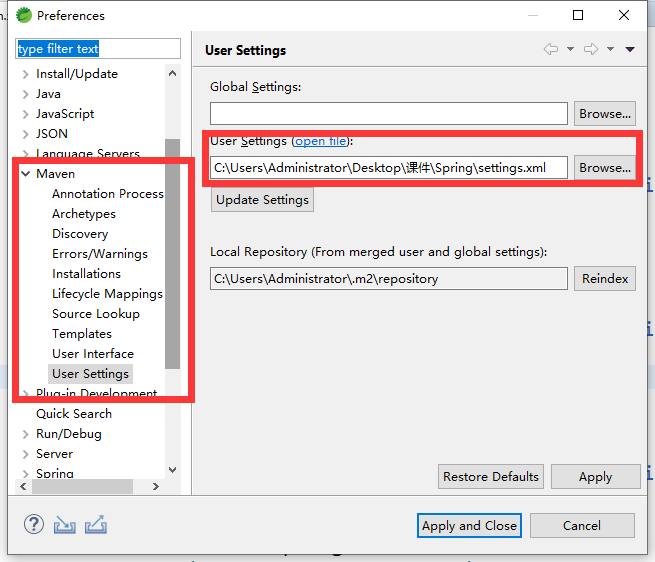
地址<https://mvnrepository.com/>

#### 使用国内镜像

创建一个maven的配置文件

参照：

<http://maven.apache.org/settings.html>



#### Pom.xml

<!-- https://mvnrepository.com/artifact/org.springframework/spring-beans -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>5.1.7.RELEASE</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.1.7.RELEASE</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-core -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>5.1.7.RELEASE</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.commons/commons-lang3 -->

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-lang3</artifactId>

<version>3.9</version>

</dependency>

## 空值注入

### Value标签

标识空值 或空字符串 “”

<property name=*"name"*><value></value></property>

### Null标签

标识Null

<property name=*"name"*><null></null></property>

## 工厂方式注入

为满足更复杂的需求，Spring也提供了工厂方式来创建更加灵活的Bean。

留意观察工厂类和实现类的创建次数

### 动态工厂 （每次都是new出来的）

抽象接口 Car

public interface Car {

public String getName();

public String getPrice();

}

实现类 BMW车

public class Bmw implements Car{

public String getName() {

// TODO Auto-generated method stub

return "别摸我";

}

public String getPrice() {

// TODO Auto-generated method stub

return "500000RMB";

}

}

汽车工厂类 CarFactory

**public** **class** CarFactory {

**public** Car getCar(String name) **throws** Exception{

**if** (name.endsWith("bmw")) {

**return** **new** Bmw();

}**else** {

**throw** **new** Exception("car not fond");

}

}

}

Bean配置

<bean id="carFactory" class="com.msb.CarFactory"></bean>

<bean id="car" factory-bean="carFactory" factory-method="getCar" >

<constructor-arg value="bmw"></constructor-arg>

</bean>

### 静态工厂

Bean配置

<bean id="carStatic" class="com.msb.CarFactoryStatic" factory-method="getCar">

<constructor-arg value="bmw"></constructor-arg>

</bean>

工厂类

**public** **class** CarFactoryStatic {

**public** **static** Car getCar(String name) **throws** Exception{

**if** (name.endsWith("bmw")) {

**return** **new** Bmw();

}**else** {

**throw** **new** Exception("car not fond");

}

}

}

## autowire自动注入

使用自动需要在配置文件中bean上添加autowire

<bean id="person" class="com.msb.Person" autowire="byName">

</bean>

<bean id="pet" class="com.msb.Pet">

<property name="name" value="kele"></property>

</bean>

实体

public class Person {

private String name;

private Pet pet;

}

public class Pet {

private String name;

}

可选两种类型

### byName

byName方式自动注入：要求注入的bean的id必须和被注入的bean对象的属性名一致

### byType

byType方式自动注入：要求注入的bean的对象类型与被注入的bean对象类型一致，并且在配置文件中的Bean相同类型必须唯一

如果存在多个，会抛异常：

No qualifying bean of type 'com.msb.Pet' available: expected single matching bean but found 2: pet,pet2

### 全局自动注入

在首行Beans标签下添加default-autowire属性。

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"

default-autowire="byType"

>

## annotation注解注入

使用注解需要导入AOP包

在配置文件中添加Context约束

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd

"

>

### <context:component-scan>

<context:component-scan base-package=*"com.msb"*></context:component-scan>

component-scan可以自动扫描包内容，并注册Bean到Spring容器

### @Component

在需要注册到容器的类上添加@Component标签，标识这个类由Spring容器接管

#### 约定大于配置

在一个类上添加@Component默认会使用首字母小写的类名作为ID注册到Spring容器。

如果需要手动指定Bean Id可以使用@Component("p")

#### 同属@Component的额外三个注解

@Controller @Service @Repository

这三个注意在MVC开发中会经常用到，除了注解名字和Component不一样之外，其余功能都一样。

Spring额外提供这三个注解的目的主要是为了区分MVC中每个类的区别。

### @Scope

使用注解注册Bean 默认的作用域还是singleton，可以使用@Scope("prototype")改变对象作用域

### @Value

在使用注解给对象注入值的时候，不再需要Get/Set方法

#### 基础类型

使用@Value注解

@Value("小明")

**private** String name;

#### 对象引用

@Autowired

**private** Pet MyPet;

使用@Autowired注解

默认是ByType的，如果需要ByName需要配合@Qualifier注解

@Autowired()

@Qualifier("p2")

**private** Pet MyPet;

## 面向切面编程 代码增强

AOP(Aspect Oriented Programming)面向切面编程。

面向切面，是与OOP(Object Oriented Programming)面向对象编程并列的编程思想。

Spring支持两种方法,那么我们在使用spring进行动态代理时究竟使用的哪一种方法呢？spring优先支持实现接口的方式,如果没有接口则使用cglib方式

### 代理

通过代理可以隐藏目标类的具体实现;在不修改目标类代码的情况下能够对其功能进行增强。

* 委托类和代理类有相同的接口或者共同的父类
* 代理类为委托类负责处理消息，并将消息转发给委托类
* 委托类和代理类对象通常存在关联关系
* 一个代理类对象与一个委托类对象关联
* 代理类本身并不是真正的实现者！而是通过调用委托类的方法来实现功能！

#### 静态代理

使用硬编码的方式增强原有方法

* 优点：可以做到不对目标对象进行修改的前提下，对目标对象进行功能的扩展和拦截。
* 缺点：因为代理对象，需要实现与目标对象一样的接口，会导致代理类十分繁多，不易维护，同时一旦接口增加方法，则目标对象和代理类都需要维护。

Girl -> 目标对象 -> 被包装/增强的对象

public class Girl implements Human{

public void eat() {

System.*out*.println("Em mmm.. mm..");

}

}

抽象接口

interface Human {

public void eat();

}

ProxyGirl 代理对象，包含对原对象方法的增强，通过构造方法传入原对象，并实现和原对象相同的接口，实现接口方法，便可以利用Java多态的特性，通过访问代理方法同时能够调起原对象的实现，并对其增强。

public class ProxyGirl implements Human {

private Human human;

public ProxyGirl() {

super();

}

public ProxyGirl(Human human) {

super();

this.human = human;

}

public void eat() {

System.*out*.println("chiqian");

human.eat();

System.*out*.println("chihou");

}

}

测试类

Girl girl = **new** Girl();

Human proxyGirl = **new** ProxyGirl(girl);

proxyGirl.eat();

#### 动态代理

动态代理是指动态的在内存中构建代理对象（需要我们制定要代理的目标对象实现的接口类型），即利用JDK的API生成指定接口的对象，也称之为JDK代理或者接口代理。

* 目标对象实现了接口 JDK动态代理
* 目标对象没有实现口CGLib【

##### JDK动态代理

##### CGLIB动态代理

##### 底层ASM

作业：

使用 Maven 构建一个 MVC的使用Spring注解的 应用程序